Problem Based Learning Report

Atharv Naphade, c3386230

SENG3160: Software Project 2: Software Implementation, Testing, and Maintenance

Dr. Mark Wallis

September 3, 2025

1. Table of Contents

2. Abstract

This is report is broken into two major sections. A literature review of software testing methodologies and a comprehensive testing plan proposal for the CoachCraft project.

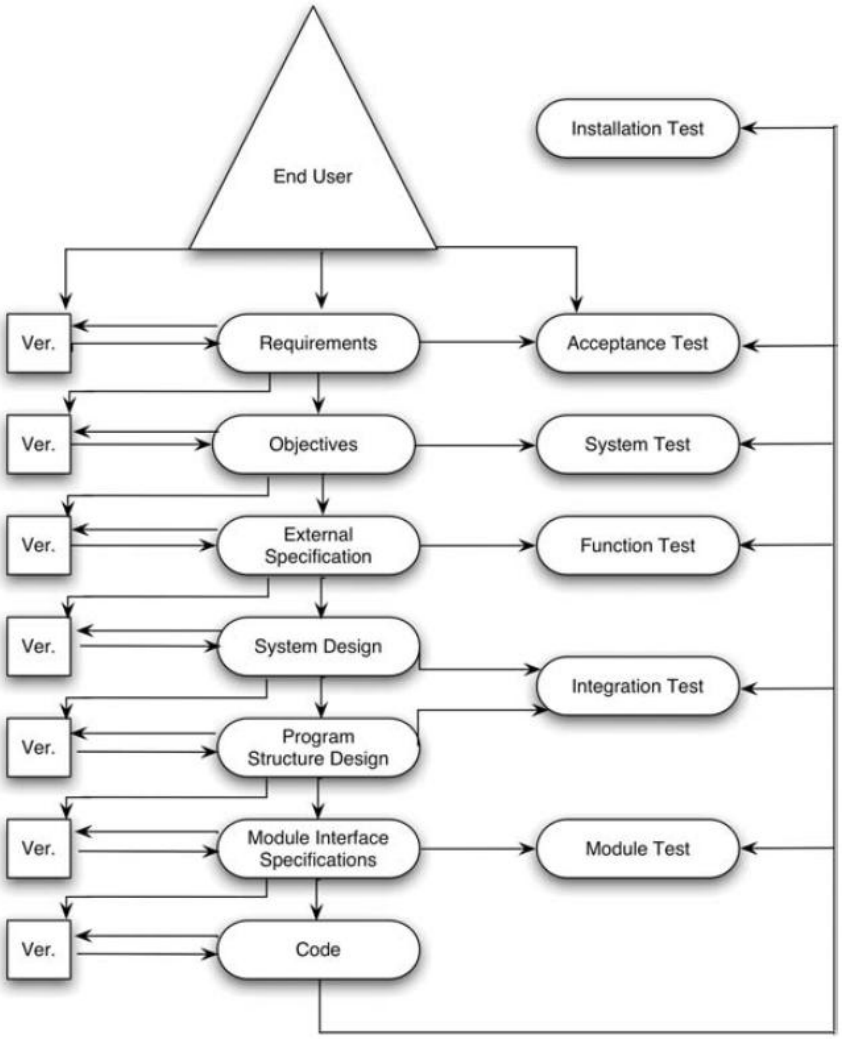
The literature review summarizes a few key software testing techniques such as unit testing, integration testing, system testing, acceptance testing, contract testing, and non-functional testing. Some information for the first section is referenced from well-regarded books including, The Art by Software Testing (2012) and Software Testing (2005), while some other information is sourced from several blogs, articles, etc. The books helped with the more traditional and well-established areas in the testing landscape including, unit and integration testing, system testing, acceptance testing. For contract testing and non-functional testing, more relevant information was found in online blogs and articles.

Building on some of these insights, the second section lays out a comprehensive testing plan for CoachCraft. The plan covers two key parts of the project, the GraphQL API and the Web SPA, which are currently in a testable state.

3. Literature Review

Figure 3.1.

Different kinds of tests shown in relation to points/artifacts in the software development life cycle in conjunction to which they are performed



Note. From The Art of Software Testing (3rd ed., p. 117), by Myers et al., 2012, John Wiley & Sons.

3.1. Introduction

Testing is an important component of software engineering as it ensures reliability, maintainability, and performance. There are various types of testing techniques and processes undertaken at different points in the software development life cycle.

For any given software project, it is important to strategically identify areas that are testable and will potentially lead to improvements, mitigations, or provide insight into the health of the project at any given point.

The following sections provide details on some of the popular types of software testing approaches.

3.2. Areas of Software Testing

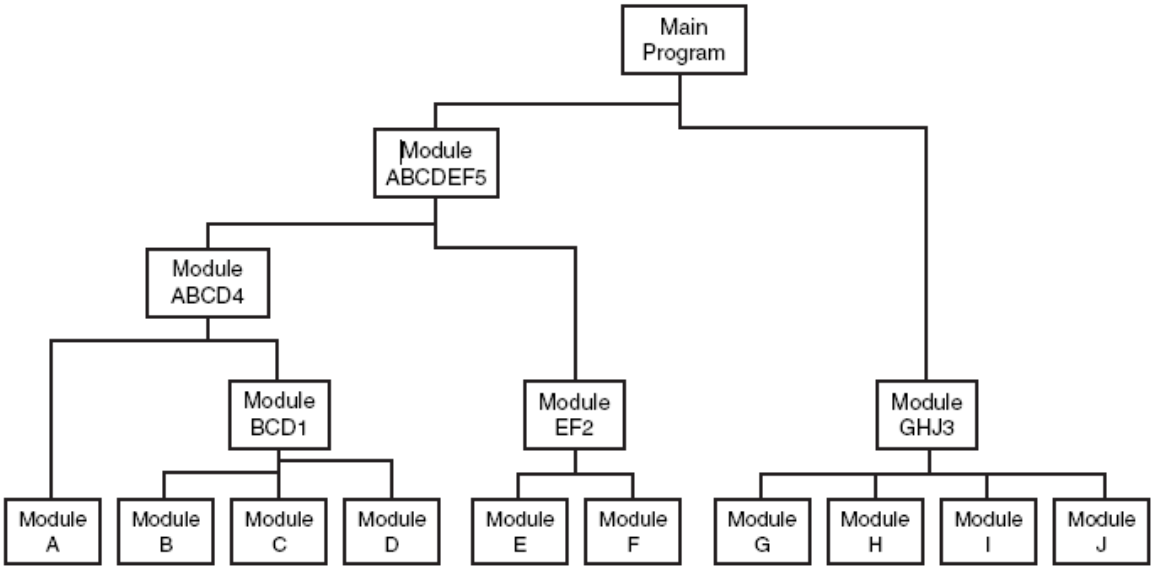
3.2.1. *Unit and Integration Testing*

Module testing (or unit testing) is a process of testing the individual sub-programs, subroutines, classes, or procedures in a program (Myers et al., 2012, p. 85). Unit testing is usually performed alongside development, this allows for quick feedback and concrete definitions for each component's behaviour such as inputs, outputs and edge cases. Myers et al., 2012 presents three motivations for why unit testing is used (p. 85) –

- It aids in managing other kinds of testing that involve multiple units working together, as it allows for working on the knowledge that each of the individual units are functioning well.
- It simplifies debugging by allowing developers to immediately direct their attention to concise units of the software program, in case of errors.
- It is a time and cost-efficient form of testing as it allows for running tests in parallel; on all units, as they are detached from each other's state.

Figure 3.2.

Representation of modules within a software program, shown as being integrated for testing.



Note. From Software Testing (2nd ed., p. 109), by R. Patton, 2005, Sams Publishing.

Integration testing is very closely linked to unit testing, as it involves testing the identified units' interactions and finding any interfacing issues. Figure 2.2. illustrates how several units/modules in a software program can be organised together in a hierarchical manner. Examples of frameworks that enable unit testing are JUnit, Vitest, pytest, etc.

3.2.2. *System Testing*

Following from unit and integration testing, system testing involves testing the proper functioning of the software program and its dependencies with all identified units running in unison. System testing also relies on a set of expected behaviours or measurable objectives (Myers et al., 2012, p. 120). However, it is important to distinguish between measurable objectives in this context and client requirements, since the latter is part of Acceptance Testing, as is made clear in Figure 2.1.

3.2.3. *Acceptance Testing*

The crucial stage in any procurement is the moment when the vendor offers the product to the buyer for inspection to determine whether the contract has been satisfied (UCC 2-606, as cited in Brannigan, 1985). This rather universal definition helps in understanding Acceptance testing fundamentally. When put in the context of software systems, acceptance testing refers to tallying the functionality of the system with the initial client requirements, as is also defined in Figure 2.1. Successful adherence to the client requirements or otherwise can be demonstrated with high-fidelity end-to-end tests using modern testing frameworks, which allow for browser automation and other platform-specific UI automation. Examples of such frameworks include Cypress, Appium, UI Automator etc.

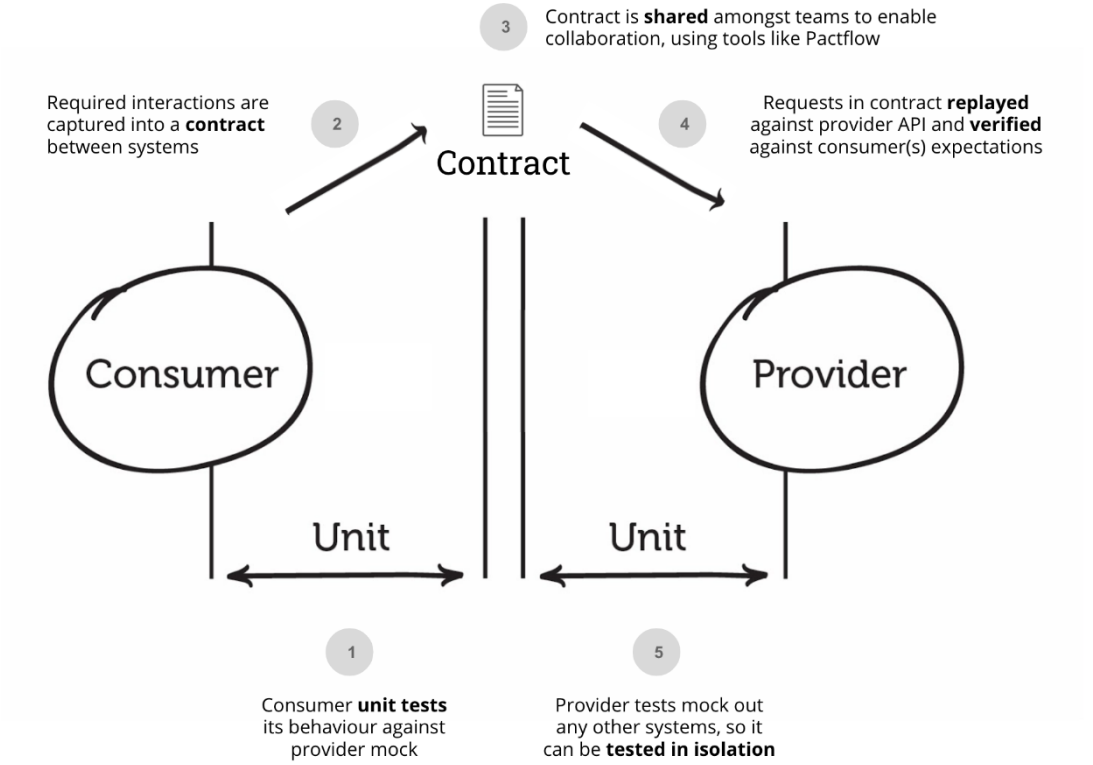
3.2.4. *Contract Testing*

Contract testing is a methodology for ensuring that two separate systems (such as two microservices) are compatible and can communicate with one another (Fellows, Sep 2023), its implementation captures interactions between two systems, storing them in a “contract”, to be used later to verify adherence. Figure 2.3. provides an overview of the contract testing implementation in Pactflow. Contract tests help ensure that consumer expectations and provider behaviour remain compatible. They are mostly used in the context of products/systems such as APIs, microservices, etc. Fellows, Jan 2023 in another blog article posted on pactflow.io, defines three strategies for contract testing with Pactflow, we will reference two relevant ones –

- Consumer driven contract testing – the consumer defines and communicates their requirements to a provider. Employed when consumers have their requirements mapped out and need the provider to conform to them.
- Provider driven contract testing – the provider defines and communicates their requirements, requiring the consumer to conform.

Figure 3.3.

Consumer-driven contract testing implementation in Pactflow.



Note. From What is Contract Testing & How is it Used? | Pactflow, by Fellows M., 2023

3.2.5. *Non-functional Testing*

Non-functional testing assesses critical aspects of a software application such as usability, performance, reliability, and security (Son, 2024). It is an umbrella category that complements functional testing and focuses on the system's overall functioning and sustainability rather than user requirements.

Son, 2024 in a blog article on non-functional testing lists 51 different types of non-functional testing, organised into 7 broad categories. Jenkins, 2019 in a blog article lists 11 categories, which are similar, referring to them as “testing parameters” in non-functional testing. Non-functional tests aim to address all these categories or factors; we will list and explain them briefly –

- Performance – speed, responsiveness and stability.
- Usability – ease of use and intuitiveness.
- Security – assessing vulnerabilities.
- Risk Assessment – evaluating potential risks associated with identified vulnerabilities.
- Reliability – ability to function reliably for extended periods of time.
- Availability – ensuring minimum downtime or meeting expected levels of uptime.
- Scalability – ability to scale with demand, including dynamic or manual scaling.
- Interoperability – ability to integrate well with other systems.
- Maintainability – ability to update and maintain with ease.
- Portability – ability to be transferred seamlessly across environments.
- Recoverability or Disaster Recovery – ability to recover quickly and fully from unanticipated events such as data loss, DDoS attacks, etc.

All the listed categories or testing factors have individual and specific testing types associated with them, we only discuss the broader categories here due to the vastness of non-functional testing.

4. Testing Plan Proposal

4.1. End-to-End tests for the Web SPA

4.1.1. *Test Environment Setup (Web E2E)*

The test environment will involve running the Web SPA in a “test mode”. This will enable a mocked substitution for AWS Cognito, disable features that are out of test scope, switch the backend URLs to a local one. Cypress Studio will be a crucial tool for developing the tests, as it allows for recording UI flows and adding assertions for E2E tests by simply interacting with the app (Cypress Documentation, 2025). These recorded UI flows are then serialised and can replayed.

This test suite uses the same isolated backend test environment described in section 3.2.1 Test Environment Setup (API), including mocked authentication, new SQLite DB per run, etc.

4.1.2. *Test Scope and Design*

The scope derives from user stories and covers the core user journeys -

- Anonymous user register and login
- CRUD Team
- CRUD Sessions
- CRUD Activities
- Navigate the SPA in both expected and unexpected manners

The test assertions for all CRUD tests will be linked to visual cues for “success”, “in progress” and “failure”. The tests will also check for consistency of the rendered UI with the internal state, data persistence on reload, offline functionality, etc.

For testing the SPA navigation, the router’s state will be tallied with the currently rendered page. It will also involve deliberate disruption attempts by sending “back”, “next” and “reload” signals to the browser when the app is idle and when it is processing, arbitrarily updating the URL bar, etc. This will be accompanied by continuously checking integrity of the state at all points.

4.2. API Testing

API Testing for CoachCraft’s GraphQL endpoints implemented with Fastify.

4.2.1. *Test Environment Setup (API)*

The testing environment setup will be fully isolated with mocked substitutes for any external services such as authentication, to a reasonable degree. The following is an implementation outline -

- For every run of the test suite, create new SQLite files/DBs from existing schemas.
- Populate test DB from dummy data, dummy data from a pre-generated dataset which should be committed to the repository.
- Implement “knobs” or configuration environment variables within the backend which will allow for easily switching to a testing mode.

- Implement a mock layer for authentication with AWS Cognito, consisting of test JWT (JSON Web Token) for authentication for the GraphQL endpoint.

4.2.2. *Designing Tests*

Designing test assertions for this test suite will consist of executing CRUD operations on all DB schema entities within scope of the end user. For each one of such entities, “happy” and “unhappy” input paths will be used (synthesized.io, 2024). “Happy” inputs will consist of generally expected, type and length accurate inputs, with assertions for expected server responses such as server response codes and data formats/structures. “Unhappy” inputs will consist of a range of unexpected inputs such as,

- Mild – wrong type, incorrect length, etc
- Moderate – null type, incorrect data format/structure, etc.
- Bad faith – “fuzzed” inputs, code injection attempts, unreasonably large inputs etc.

4.2.3. *Tooling*

A set of off-the-shelf tools and frameworks’ features will be used for implementing the proposed test suite.

- `fastify.inject()` for simulating HTTP requests to the GraphQL endpoint (fastify.dev, n.d.). It allows for testing endpoints without starting up the HTTP server, easing setup.

- Vitest for assertions, structured test status logs, parallelism support for fast execution (vitest.dev, 2025).
- GitHub Actions for executing the test suite on various triggers such as on creating PRs, merges into the main branch, etc.

5. Conclusion

Despite the number of references used in this report, from a book going back to 2005 to very recent blogs, we found that the overall testing landscape remains fragmented and opinionated. There are certain types of testing that have in-fact been standardised in industry with many different implementations across languages. A disarray of terminology and even concepts is apparent throughout the references in this report.

In online discourse there seems to be an air of uncertainty as to the usefulness of certain parts of the field. This includes end-to-end tests for frontends, test driven development for unit testing, etc. Choosing the right tests for the right components of a system remains to be quite challenging. The landscape also appears to be scattered with instances of diminishing returns for the aforementioned areas.

Despite all mentioned problems, there exist some fascinating frameworks implemented around some of these methodologies and theories, such as, Puppeteer, Cypress, Vitest, etc with many more for other languages.

6. Bibliography

Brannigan, V. (1985). Acceptance Testing - The Critical Problem in Software Acquisition. *IEEE Transactions on Biomedical Engineering, 1985-04, Vol.BME-32 (4), p.295-299*.

Cypress Documentation. (2025, August 22). *Cypress Studio: Build Tests by Interacting with Your App | Cypress Documentation*. https://docs.cypress.io/app/guides/cypress-studio

fastify.dev. (n.d.). *Testing | Fastify*. Retrieved September 2, 2025, from https://fastify.dev/docs/latest/Guides/Testing/

Fellows, M. (2023a, January 5). *Contract Testing vs. Schema Testing | Pactflow*. https://pactflow.io/blog/contract-testing-using-json-schemas-and-open-api-part-1/

Fellows, M. (2023b, September 2). *What is Contract Testing & How is it Used? | Pactflow*. https://pactflow.io/blog/what-is-contract-testing/

Jenkins, S. (2019, June 26). *A Guide to Non-Functional Requirements - OctoPerf*. https://blog.octoperf.com/a-guide-to-non-functional-requirements/

Myers, G. J. ., Sandler, Corey., & Badgett, Tom. (2012). The Art of Software Testing. In *The Art of Software Testing* (3rd ed.). John Wiley & Sons.

Patton, Ron. (2005). Software Testing. In *Software Testing* (2nd ed.). Sams Publishing.

Son, H. (2024, September 14). *Complete Guide to Non-Functional Testing: 51 Types, Examples & Applications - TestRail*. https://www.testrail.com/blog/non-functional-testing/

synthesized.io. (2024, September 12). *Balancing Happy Paths, Golden Paths, and Negative Testing with AI-Driven Test Data Management - Blog - Synthesized*. https://www.synthesized.io/post/balancing-happy-path-negative-testing

vitest.dev. (2025, May 17). *Parallelism | Guide | Vitest*. https://vitest.dev/guide/parallelism